SOEN2070 C++程序设计

08 算法

刘安 anliu@suda.edu.cn 2023-2024-2

标准库算法

https://en.cppreference.com/w/cpp/algorithm

- 算法库定义了一系列算法：搜索、排序、计数、操作等
- 算法作用于一个范围，即一个元素序列，由[first, last)表示
- first表示序列中的第一个元素，last表示序列中的尾后元素

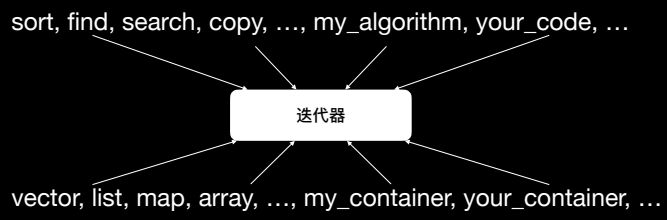
C++ **Algorithm library**

Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as [`first` , `last`) where `last` refers to the element *past* the last element to inspect or modify.

算法、容器与迭代器

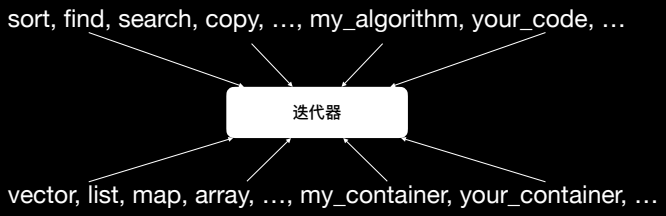
- 使用迭代器来连接算法和数据
- 算法使用者了解迭代器的使用方法，但不需要了解迭代器实际如何访问数据
- 数据提供者向用户提供相应的迭代器，而不是数据存储的细节信息



算法、容器与迭代器

The reason STL algorithms and containers work so well together is that they don't know anything about each other.

— Alex Stepanov, C++标准模板库的主要设计者和实现者



简单搜索算法find

简单搜索算法find

```
template<typename In, typename T>
//要求In是输入迭代器，T可以通过==来进行比较
In find(In first, In last, const T& val)
{
    while (first != last && *first != val) ++first;
    return first;
}
```


使用find

```
void f(vector<int>& v, int x)
{
    auto p { find(v.begin(), v.end(), x) };
    if (p != v.end()) {
        //在v中找到了x
    }
    else {
        //v中没有x
    }
}
```

注意函数和g形参的区别

```
template<typename In, typename T>
//要求In是输入迭代器，T可以通过==来进行比较
In find(In first, In last, const T& val)
{
    while (first != last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x)
{
    vector<int>::iterator p { find(v.begin(), v.end(), x) };
    if (p != v.end()) { /*在v中找到了x*/ }
    else { /*v中没有x*/ }
}

void g(list<string>& v, string x)
{
    list<string>::iterator p { find(v.begin(), v.end(), x) };
    if (p != v.end()) { /*在v中找到了x*/ }
    else { /*v中没有x*/ }
}
```


注意函数和g形参的区别


```
template<typename In, typename T>
//要求In是输入迭代器，T可以通过==来进行比较
In find(In first, In last, const T& val)
{
    while (first != last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x)
{
    vector<int>::iterator p { find(v.begin(), v.end(), x) };
    if (p != v.end()) { /*在v中找到了x*/ }
    else { /*v中没有x*/ }
}

void g(list<string>& v, string x)
{
    list<string>::iterator p { find(v.begin(), v.end(), x) };
    if (p != v.end()) { /*在v中找到了x*/ }
    else { /*v中没有x*/ }
}
```


通用搜索算法find_if

通用搜索算法find_if

```
template<typename In, typename Pred>
//要求In是输入迭代器
//可以用一个类型为T的实参调用Pred，得到一个bool值
//其中类型T是迭代器指向的值类型
In find_if(In first, In last, Pred pred)
{
    while (first != last && !pred(*first)) ++first;
    return first;
}
```

谓词
返回bool值的函数

使用find_if - 找到第一个奇数

```
bool odd(int x) { return x % 2; }

void f(vector<int>& v)
{
    auto p { find_if(v.begin(), v.end(), odd) };
    if (p != v.end()) { /*在v中找到了一个奇数*/ }
    else { /*v中没有奇数*/ }
}
```

这里是函数名
不是函数调用

使用find_if - 找到第一个大于42的数

```
bool larger_than_42(double x) { return x > 42; }

void f(list<double>& v)
{
    auto p { find_if(v.begin(), v.end(), larger_than_42) };
    if (p != v.end()) { /*在v中找到了第一个大于42的数*/ }
    else { /*v中没有大于42的数*/ }
}
```

如何找到第一个大于41的数?　　如何找到第一个大于19的数?

使用find_if - 找到第一个大于y的数

```
bool larger_than_42(double x) { return x > 42; }
bool larger_than_41(double x) { return x > 41; }
bool larger_than_19(double x) { return x > 19; }

void f(list<double>& v)
{
    auto p { find_if(v.begin(), v.end(), larger_than_42) };
    if (p != v.end()) { /*在v中找到了第一个大于42的数*/ }
    else { /*v中没有大于42的数*/ }
}

bool larger_than_y(double x, double y) { return x > y; }
```


find_if要求pred只能有一个参数

```
template<typename In, typename Pred>
//要求In是输入迭代器,
//可以用一个类型为T的实参调用Pred, 得到一个bool值
//其中类型T是迭代器指向的值类型
In find_if(In first, In last, Pred pred)
{
    while (first != last && !pred(*first)) ++first;
    return first;
}

bool larger_than_y(double x, double y) { return x > y; }
```

只有一个参数

使用全局变量让y成为谓词的隐含参数

```
double y; //全局变量，这样可以在larger_than_y函数中使用y
bool larger_than_y(double x) { return x > y; }

void f(list<double>& v)
{
    y = 41;
    auto p { find_if(v.begin(), v.end(), larger_than_y) };
    if (p != v.end()) { /*在v中找到了第一个大于41的数*/ }

    y = 19;
    auto q { find_if(v.begin(), v.end(), larger_than_y) };
    if (q != v.end()) { /*在v中找到了第一个大于19的数*/ }
}
```


我们期望的Larger_than_y

- Larger_than能够作为谓词被调用，比如pred(*first)
- Larger_than能否存储一个值，比如41或y，以备调用时使用

```
void f(list<double>& v, int y)
{
    auto p { find_if(v.begin(), v.end(), Larger_than(41)) };
    if (p != v.end()) { /*在v中找到了第一个大于41的数*/ }

    auto q { find_if(v.begin(), v.end(), Larger_than(y)) };
    if (q != v.end()) { /*在v中找到了第一个大于y的数*/ }
}
```


函数对象

函数对象 - 具有函数调用运算符( )的类

- Larger_than能够作为谓词被调用，比如pred(*first)
- Larger_than能否存储一个值，比如41或y，以备调用时使用
- 类Larger_than能够满足上述两个条件

成员y存储一个值，并在构造函数中通过参数传递该值

```
class Larger_than {
    int y;
public:
    Larger_than(int yy) : y { yy } {}
    bool operator()(int x) const { return x > y; }
};
```

重载函数调用运算符( )使得类Larger_than可以被调用

```
auto p { find_if(v.begin(), v.end(), Larger_than(41)) };

template<typename In, typename Pred>
In find_if(In first, In last, Pred pred)

pred(*first)

class Larger_than {
    int y;
public:
    Larger_than(int yy) : y { yy } {}
    bool operator()(int x) const { return x > y; }
};
```

构造函数，y的值为41

pred是一个Larger_than的对象

()是函数调用运算符
实参*first传递给形参x

lambda表达式

lambda表达式

- 一种简写语法：定义一个函数对象类型然后立即创建该类型的一个对象
- 语法：[捕获列表] (参数列表) -> 返回类型 {函数体}
  - 捕获列表：捕获lambda表达式所在函数中使用的局部变量，通常为空
  - 可以忽略参数列表和返回类型，但必须包括捕获列表和函数体
    - 省略括号()和参数列表等价于指定一个空参数列表
    - 忽略返回类型，编译器根据返回的表达式类型来确定

```
auto f = [] { return 42; };
cout << f() << endl;
```


lambda表达式 [捕获列表] (参数列表) -> 返回类型 {函数体}

```
void func(int y)
{
    auto f = [] (double x) { return x > 31; };
    cout << f(30) << endl; // 0
    cout << f(40) << endl; // 1

    // y是lambda表达式所在函数的局部变量
    // 放在捕获列表中后，可以在lambda表达式的函数体中使用
    auto g = [y] (double x) { return x > y; };
}
```

lambda表达式 [捕获列表] (参数列表) -> 返回类型 {函数体}

```
void func(list<double>& v, int y)
{
    auto p { find_if(v.begin(), v.end(), [](double x) { return x > 31; }) };
    if (p != v.end()) { /*在v中找到了第一个大于31的值*/ }

    auto q { find_if(v.begin(), v.end(), [y](double x) { return x > y; }) };
    if (q != v.end()) { /*在v中找到了第一个大于y的值*/ }
}
```


值捕获与引用捕获

- 值捕获：拷贝变量的值
  - 在lambda创建时拷贝，而不是调用时拷贝

```
void func()
{
    int v { 42 };
    auto f { [v] { return v; } }; // 创建lambda，拷贝v的值
    cout << f() << endl; // 42
    v = 0; // 修改v的值不影响lambda中的v
    cout << f() << endl; // 42
}
```


值捕获与引用捕获

- 引用捕获：在lambda函数体内使用该变量时，实际使用该引用绑定的对象

```
void func()
{
    int v { 42 };
    auto f { [&v] { return v; } }; // 创建lambda，引用捕获
    cout << f() << endl; // 42
    v = 0; // 修改v的值有影响，因为lambda通过引用使用对象v
    cout << f() << endl; // 0
}
```


值捕获与引用捕获

```
set<string> teas { "black", "green", "oolong" };
string banned { “boba" }; // this is not a tea
auto liked_by_alice = [&teas, banned] (auto type) {
    return teas.count(type) && type != banned;
};
```

隐式捕获（教材352页，表10.1）

- 编译器根据lambda函数体代码来推断要使用哪些变量
- &表示采用引用捕获，=表示采用值捕获

```
set<string> teas { "black", "green", "oolong" };
string banned { “boba" }; // this is not a tea
//capture all by value, except teas is by reference
auto f { [=, &teas] (auto type) {
    return teas.count(type) && type != banned; }};

//capture all by referecne, except banned is by value
auto g { [&, banned] (auto type) {
    return teas.count(type) && type != banned; }};
```


数值算法

STL标准库中的四种数值算法 #include <numeric>

x = accumulate(b, e, i)	累加序列中的值，比如对{a,b,c,d}计算i+a+b+c+d 结果x的类型与初始值i的类型一致
x = inner_product(b, e, b2, i)	将两个序列中的对应元素相乘并将结果累加，比如对{a,b,c,d}和{e,f,g,h}计算i+ae+bf+cg+dh 结果x的类型与初始值i的类型一致
r = partial_sum(b, e, r)	对一个序列的前n个元素进行累加，并将累加结果生成一个序列，比如对{a,b,c}生成{a,a+b,a+b+c}
r = adjacent_difference(b, e, b2, r)	对一个序列的相邻元素进行减操作，并将得到的差生成一个序列，比如对{a,b,c,d}生成{a,b-a,c-b,d-c}

累加算法accumulate

累加算法accumulate

```
template<typename In, typename T>
//要求In是输入迭代器，其指向的值类型为T
//T类似数字，支持+，-，*，/
T accumulate(In first, In last, T init)
{
    while (first != last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```


使用accumulate

```
void f(vector<double>& vd, int* p, int n) //p指向某个数组首地址，n数组长度
{
    double sum { accumulate(vd.begin(), vd.end(), 0.0) }; //0.0而不是0
    int sum2 { accumulate(p, p + n, 0) };

    double s1 { 0 };
    s1 = accumulate(vd.begin(), vd.end(), s1); //ok
    int s2 {accumulate(vd.begin(), vd.end(), s2)}; //bad
    float s3 { 0 };
    accumulate(vd.begin(), vd.end(), s3); //bad
}
```

Variable 's2' is uninitialized when used within its own initialization

泛化累加算法accumulate

```
template<typename In, typename T, typename BinOp>
//要求In是输入迭代器，其指向的值类型为T
//T类似数字，支持+，-，*，/
//BinOp可以对两个T执行一个操作
T accumulate(In first, In last, T init, BinOp op)
{
    while (first != last) {
        init = op(init, *first);
        ++first;
    }
    return init;
}

vector<double> a { 1.1, 2.2, 3.3, 4.4 };
double s { accumulate(a.begin(), a.end(), 1.0, multiplies<double>()) };
cout << s << endl; //35.1384 = 1.0 * 1.1 * 2.2 * 3.3 * 4.4
```

标准库函数对象
在<functional>中定义
比如plus, minus, divides, modulus

给定物品的单位价格和单位数，计算所有物品的总价格

```
struct Record {
    double unit_price; //单位价格
    int units; //销售的单位数
};

void f()
{
    vector<Record> items { {1.1, 10}, {2.2, 20},
                           {3.3, 30}, {4.4, 40} };
    double price { accumulate(items.begin(), items.end(), 0.0,
                   [] (double v, const Record& r)
                    { return v + r.unit_price * r.units; })
                 };
    cout << price << endl; // 330 = 0 + 11 + 44 + 99 + 176
}
```

内积算法inner_product

内积算法inner_product

```
template<typename In, typename In2, typename T>
//要求In和In2是输入迭代器，其指向的值类型为T
T inner_prodct(In first, In last, In2 first2, T init)
{
    while (first != last) {
        init = init + (*first) * (*first2);
        ++first;
        ++first2;
    }
    return init;
}
```


泛化内积算法inner_product

```
template<typename In, typename In2, typename T, typename BinOp,
typename BinOp2>
//要求In和In2是输入迭代器，其指向的值类型为T
//T类似数字，支持+，-，*，/
//BinOp和BinOp2均可以对两个T执行一个操作
T inner_prodct(In first, In last, In2 first2, T init, BinOp op,
BinOp2 op2)
{
    while (first != last) {
        init = op(init, op2(*first, *first2));
        ++first;
        ++first2;
    }
    return init;
}
```


拷贝算法

STL标准库中的三种拷贝算法 #include <algorithm>

copy(b, e, b2)	将[b:e)拷贝到[b2:b2+(e-b))
unique_copy(b, e, b2)	将[b:e)拷贝到[b2:b2+(e-b))，禁止拷贝相邻的相同元素
copy_if(b, e, b2, p)	将[b:e)拷贝到[b2:b2+(e-b))，但是仅拷贝满足谓词p的元素

拷贝算法copy

拷贝算法copy

- 输入序列类型与输出序列类型可以不同
- 自行确保输出序列有足够大的空间来保存拷贝来的元素

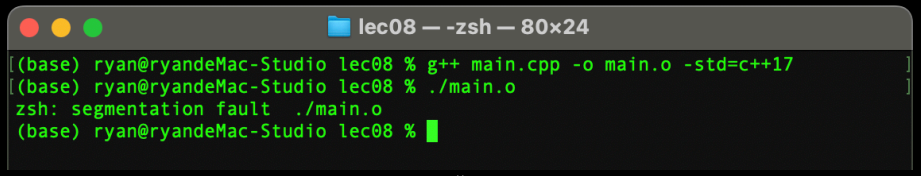
```
template<typename In, typename Out>
//要求In是输入迭代器，Out是输出迭代器
Out copy(In first, In last, Out res)
{
    while (first != last) {
        *res = *first;
        ++res;
        ++first;
    }
    return res;
}
```


拷贝算法copy

- 输入序列类型与输出序列类型可以不同
- 自行确保输出序列有足够大的空间来保存拷贝来的元素

```
list<int> li { 1, 2, 3, 4, 5, 6, 7};
vector<double> vd;
copy(li.begin(), li.end(), vd.begin());
```

vd是一个空的vector
没有空间来存储拷贝的元素

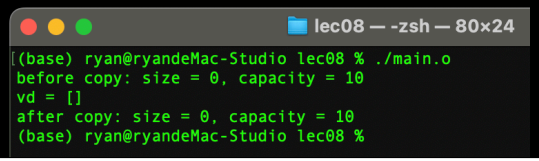


拷贝算法copy

- 输入序列类型与输出序列类型可以不同
- 自行确保输出序列有足够大的空间来保存拷贝来的元素

```
list<int> li { 1, 2, 3, 4, 5, 6, 7};
vector<double> vd;
vd.reserve(10);
copy(li.begin(), li.end(), vd.begin());
```

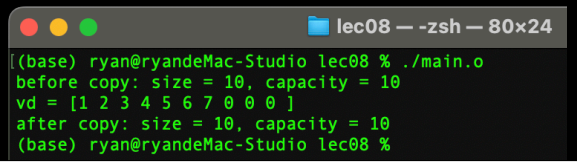
vd仍然是一个空的vector
此时begin迭代器等于end迭代器
不指向有效存储空间



拷贝算法copy

- 输入序列类型与输出序列类型可以不同
- 自行确保输出序列有足够大的空间来保存拷贝来的元素

```
list<int> li { 1, 2, 3, 4, 5, 6, 7};
vector<double> vd;
vd.resize(10);
copy(li.begin(), li.end(), vd.begin());
```



有条件拷贝算法copy_if

```
template<typename In, typename Out, typename Pred>
//要求In是输入迭代器，Out是输出迭代器
//可以用一个类型为T的实参调用P，得到一个bool值
//其中类型T是迭代器In指向的值类型
Out copy_if(In first, In last, Out res, Pred p)
{
    while (first != last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

只拷贝令谓词为真的元素

拷贝值大于42的所有元素

```
list<int> li { 10, 20, 30, 40, 50, 60, 70, 80};
vector<double> vd;
vd.resize(li.size());
copy_if(li.begin(), li.end(), vd.begin(), Larger_than(42));

class Larger_than {
    int y;
public:
    Larger_than(int yy) : y { yy } {}
    bool operator()(int x) const { return x > y; }
};
```

排序算法

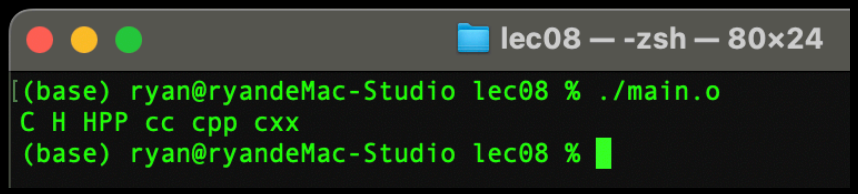
排序算法sort

```
template<typename Ran>
//要求Ran是随机访问迭代器
void sort(Ran first, Ran last);

template<typename Ran, typename Cmp>
//要求Ran是随机访问迭代器
//可以用Cmp而不是<比较两个T是否前者小于后者，得到一个bool值
//其中类型T是迭代器Ran指向的值类型
void sort(Ran first, Ran last, Cmp cmp);
```


字符串排序

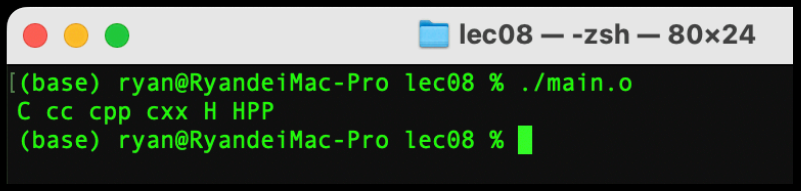
```
vector<string> v { "H", "HPP", "cpp", "cc", "C", "cxx" };
sort(v.begin(), v.end());
cout << v << endl;
```



默认是大小写敏感的字典顺序，该顺序的详细定义见教材80页

不考虑大小写的字符串排序

```
vector<string> v { "H", "HPP", "cpp", "cc", "C", "cxx" };
sort(v.begin(), v.end(), No_case());
cout << v << endl;
```

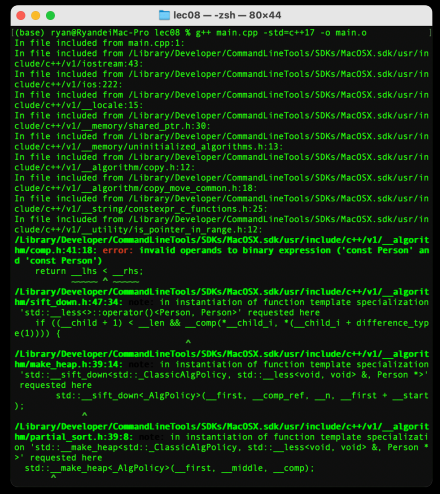


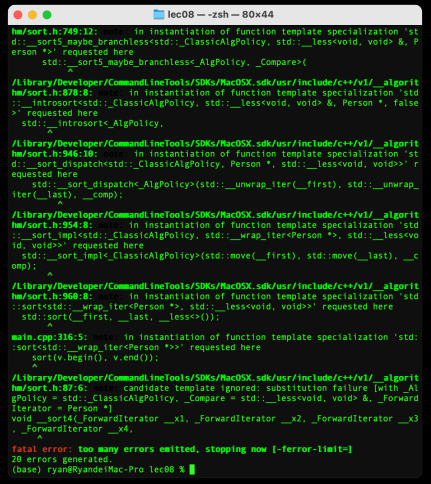
```cpp
struct No_case {
    bool operator()(const string& x, const string& y) const
    {
        for (int i { 0 }; i < x.length(); ++i) {
            if (i == y.length()) return false; // y < x
            // tolower() returns int
            char xx { static_cast<char>(tolower(x[i])) };
            char yy { static_cast<char>(tolower(y[i])) };
            if (xx < yy) return true; // x < y
            if (yy < xx) return false; // y < x
        }
        if (x.length() == y.length()) return false; // x == y
        return true; // x < y since x.length() < y.length()
    }
};
```


员工排序

```cpp
struct Person {
    string name;
    int age;
    double salary;
};

vector<Person> v { {"Alice", 20, 1000}, /*…*/ {"Jack", 60, 6000} };
sort(v.begin(), v.end());
cout << v << endl;
```



sort需要比较两个Person对象的大小，如何比较?

- 方法一：为Person重载运算符<

```cpp
//以年龄为主排序，如果年龄相等，根据收入排序
bool operator<(const Person& a, const Person& b)
{
    if (a.age == b.age) return a.salary < b.salary;
    return a.age < b.age;
}
```

sort需要比较两个Person对象的大小，如何比较?

- 方法一：为Person重载运算符<
- 方法二：为sort函数提供第三个参数，一个可以比较两个Person对象的谓词

```
sort(v.begin(), v.end(), []
    (const Person&a, const Person& b) {
        //以收入为主排序，如果收入相等，根据年龄排序
        if (a.salary == b.salary) return a.age < b.age;
        return a.salary < b.salary;
    }
);
```


二分搜索算法

二分搜索算法binary_search

```
template<typename Ran, typename T>
//要求Ran是随机访问迭代器，类型T是迭代器Ran指向的值类型
//要求[first, last）序列已经有序
bool binary_search(Ran first, Ran last, const T& val);

template<typename Ran, typename T, typename Cmp>
//要求Ran是随机访问迭代器
//可以用Cmp而不是<比较两个T是否前者小于后者，得到一个bool值
//其中类型T是迭代器Ran指向的值类型
//要求[first, last）序列已经有序
bool binary_search(Ran first, Ran last, const T& val, Cmp cmp);
```

binary_search()返回bool值，关注某个值是否在给定的序列中

equal_range

- equal_range返回一个std::pair
  - 包含两个前向迭代器，用来指示元素等于value的区间

std::equal_range

Defined in header <algorithm>

```
template< class ForwardIt, class T >
std::pair<ForwardIt, ForwardIt>
    equal_range( ForwardIt first, ForwardIt last, const T& value );
```

```
vector<int> v { 2, 4, 5, 6, 6, 9 };
auto p { equal_range(v.begin(), v.end(), 6) };
for (auto iter { p.first }; iter != p.second; ++iter)
    cout << *iter << endl;
```

lower_bound

- lower_bound返回一个前向迭代器it，它指向第一个不小于value的元素

std::lower_bound

Defined in header <algorithm>

```
template< class ForwardIt, class T >
ForwardIt lower_bound( ForwardIt first, ForwardIt last,
                       const T& value );
```

```
vector<int> v { 2, 4, 5, 6, 6, 9 };
auto it { lower_bound(v.begin(), v.end(), 5) };
cout << *it << endl; // 5
```


upper_bound

- upper_bound返回一个前向迭代器it，它指向第一个大于value的元素

std::upper_bound

Defined in header <algorithm>

```
template< class ForwardIt, class T >
ForwardIt upper_bound( ForwardIt first, ForwardIt last,
                       const T& value );
```

```
vector<int> v { 2, 4, 5, 6, 6, 9 };
auto it { upper_bound(v.begin(), v.end(), 5) };
cout << *it << endl; // 6
```


计算年龄大于等于x且小于等于y的人的平均工资

```
//计算年龄大于等于x且小于等于y的人的平均工资
double avg_salary(const vector<Person>& v, int x, int y)
{
    double total_salary { 0.0 };
    int count { 0 };

    for (const auto& person : v) {
        if (person.age >= x && person.age <= y) {
            total_salary += person.salary;
            ++count;
        }
    }

    if (count == 0) return 0;
    return total_salary / count;
}
```


计算年龄大于等于x且小于等于y的人的平均工资

```
//注意Person类重载了运算符<，以年龄为主收入为辅进行排序，比较的是两个Person类对象
double avg_salary(const vector<Person>& v, int x, int y)
{
    auto first { lower_bound(v.begin(), v.end(), Person {"no_name", x, 0}) };
    auto last  { upper_bound(v.begin(), v.end(), Person {"no_name", y, 0}) };
    auto n { distance(first, last) };
    if (n == 0) return 0;
    double s { accumulate(first, last, 0.0, []
        (double v, const Person& p)
        { return v + p.salary; }) };
    return s / n;
}
```

value是一个Person类对象

std::distance

Defined in header <iterator>

```
template< class InputIt >
typename std::iterator_traits<InputIt>::difference_type    (constexpr since C++17)
    distance( InputIt first, InputIt last );
```

Returns the number of hops from `first` to `last`.